*Carnegie Mellon University, School of Computer Science*
*Master of Software Engineering*

# Evaluation of AgitarOne

## Analysis of Software Artifacts
### Final Project Report
### April 24, 2007

**Edited for public release by Jonathan Aldrich.**

**Team members**
Tadashi Tsuji
Ayo Akinyele
Fabian Hueppi
Ed Yampratoom
Irandi Upeka Bulumulla

# Table of Contents

# Introduction

### Motivation for AgitarOne

Writing unit test cases manually is a time-consuming task. The test cases that we have written may not cover all invalid, unexpected inputs or call sequences. Moreover, inspecting code for compliance and bugs is also time-consuming. Finally, collecting and presenting test data is a tedious and error prone task that should be automated.

### Description of AgitarOne

AgitarOne is a comprehensive and integrated unit testing tool for Java. The main feature of the tool is that it automatically generates JUnit tests that provide good code coverage. These tests can complement hand-written tests that require deep application knowledge.

AgitarOne includes a unique tool called Agitator which exercise code by testing them with a wide range of inputs and observe the code's behavior. Agitator automatically generates test data to exercise the code with a high level of coverage. Agitation results are presented as a short list of observations, which are reviewed by the developer. These observations can be turned into assertions that form the basis of regression tests for the code.

AgitarOne also includes a set of code rules that can automatically detect common coding errors and standards violation. These rules can be created and customized to suit the project or organization's requirements.

As a management tool, AgitarOne provides a project dashboard which summarizes the project testing status at a glance. The project summary shows complexity, usage, coverage, and test status over time. It also identifies ownership of classes.

### Code Bases

We will evaluate AgitarOne by running it on a number of code bases.

- **Serendipity's code base:** This is the sensor placement configuration tool developed by Serendipity team for last year's Studio project that we have to enhance.
- **Crystal Analysis**
- **DWP:** A static analysis tool built on Crystal.
- **Board Game:** The code provided as a test bed for our assignment.

# JUnit Test Generation

AgitarOne's test generation features can be used as regression suites for legacy code or for developers to learn about their code. In this experiment we evaluate the effectiveness of AgitarOne for regression testing.

### Legacy code

AgitarOne is ideal for performing regression testing on legacy code. The generated tests can provide a safety net that developers can use to ensure that their refactoring activities don't have unintended side effects.

The first step for using AgitarOne on legacy code is to generate the tests on the existing code. As mentioned, in a legacy code scenario the intent of the generated tests is to prevent breaking the original behavior of the code. So AgitarOne assumes that the

legacy code is correct and thus creates tests by inspecting the functionality of the code. Therefore, all test cases pass. These tests can be best described as characterization tests[1]. They are based not on what the code is supposed to do, but what it actually does.

After the tests are generated, the programmer can start with refactoring and modifying the original code base. The generated tests should be rerun after every modification to check if the tests fail. These generated tests can also be included into a regression test suite.

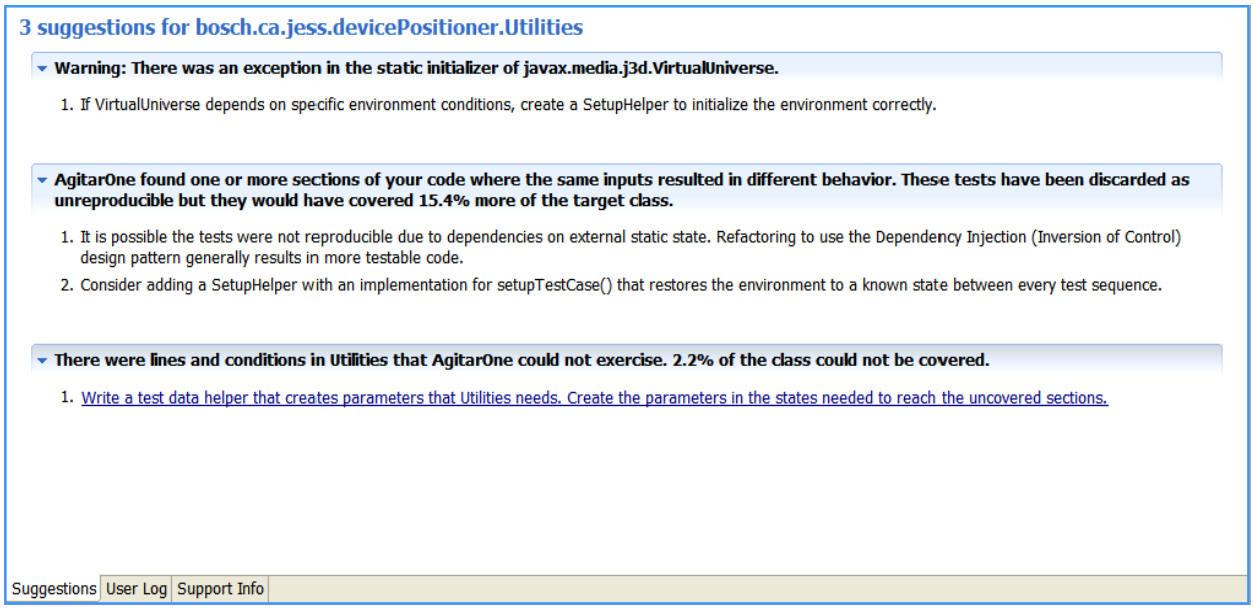


**Figure 1: If AgitarOne can't cover all the code, it provides advice on how to help improving the coverage.**

[1] http://en.wikipedia.org/wiki/Characterization_Test

## Experiment description

The objective of this experiment was to evaluate how effective AgitarOne is at catching bugs that are introduced into Serendipity's code base. In this experiment, we used AgitarOne as is recommended for legacy code and made no modifications to the test cases.

We ran the test generation for two classes in the code base. These two classes are described in the tables below.

| **Name** | **bosch.ca.jess.devicePositioner.Utilities.java** |
|---|---|
| **Description** | This class does calculations in a 3D coordination system. For example it has methods that calculate the new location of a point if it is moved a certain distance with a certain angle. |
| **Methods in original class:** | 10 |
| **Generated test methods:** | 47 |
| **Initial test coverage:** | 93% |
| **Comment:** | Since the original code does many calculations, the bugs we introduced were mostly about tampering with some of the intermediate values in the calculation: |

**Table 1: Data of Utilities.java**

| **Name** | **bosch.ca.jess.JessManager.java** |
|---|---|
| **Description** | This class manages all operations related to Jess. For example, it has methods for registering objects in Jess, managing the rule engine's focus when it is running etc. |
| **Methods in original class:** | 20 |
| **Generated test methods:** | 7 |
| **Initial test coverage:** | 37% |

**Table 2: Data of JessManager.java**

## Results of the experiment

We introduced the following bugs into the code and got the following results.

**bosch.ca.jess.devicePositioner.Utilities.java**

| | |
|---|---|
| **Bug 1** | In a method that calculates the slope between two points, we just added a random number to the result.<br><br>**Result:** 6 tests failed<br>This method was used in many of the other test methods and thus the change was reflected in many failures. |
| **Bug 2** | One method needs to find the room in which an object is located. For this the method accepts the object as a parameter and returns the room object. We changed the method so that it returns just a new room object.<br><br>**Result:** 1 test failed |
| **Bug 3** | In another test we just changed a call from Math.tan(param) to Math.cos(param) in one of the branches of an if statement.<br><br>**Result:** 3 tests failed |
| **Bug 4** | One method returns a Point object. We tried to switch the X and Y coordinates of this return value.<br><br>**Result:** 1 test failed |
| **Bug 5** | We changed the calculation so that the Z value of the return object from Bug 4 deviates from its intended value.<br><br>**Result:** Bug wasn't found.<br>**Reason:** For all the methods that did calculations on a point object, the generated tests only asserted the X value. Any changes on other members of the object weren't reflected in failed tests. |

**Table 3: Bugs in Utilities.java**

**bosch.ca.jess.JessManager.java**

| | |
|---|---|
| **Bug 1** | JessManager is a singleton class, and therefore should only be initialized through the *getInstance()* method. We changed this so that a new instance of JessManager is created each time the constructor is called.<br><br>**Result:** Bug wasn't found |
| **Bug 2** | If *intializeEngine()* method is run more than once it will cause an exception. As such, *intializeEngine()* is run from inside the constructor only. We changed the code so that this is no longer called from within the constructor.<br><br>**Result:** 1 test failed<br>Agitar was able to identify this and generate a negative test case for it. |

**Table 4: Bugs in JessManager.java**

## Evaluation

**Benefits**

- **Exploring branches:** From the tests for Utilities class, we can summarize that the Agitar tests could find changes in the mathematical calculations even if they were in nested if statements. Agitar created enough tests to cover most of the branches in a method.

- **Negative test cases:** In Bug 2 for our JessManager class, AgitarOne was able to identify that the initializeEngine() method should not run more than once. Therefore, it created a negative test case which ran this method twice.

- **Save resources:** AgitarOne can be very beneficial because creating a regression test suite manually would take up much more resources.

- **No false positives:** Since the tool checks whether results match their expectations, it is unlikely that tests will fail, if the behavior of the code didn't change.

- **Small learning curve:** It needs little knowledge to start using the tool, and quickly creates a good baseline for the testing. Also the creation of the helper classes does not seem to be difficult and AgitarOne already provides a wizard for creating them.

**Limitations**

- **Handling complex return objects:** The problem was that for methods that returned a complex object, only certain members of that object were tested for correctness (see Bug 5 in the Utilities class). This can result in many false negatives.

- **Non deterministic number of test cases:** Upon regeneration of test cases for the same class, we received files containing different numbers of test cases. This is a disadvantage because the optimized coverage may not be achieved on the first generation of test cases. The user might have to generate the test cases several times and may not know if the next generation will achieve a higher coverage than the current one.

- **Usability for new user:** Newly generated tests override the existing tests. This might be cumbersome for new users, since they have to remember to save their existing test cases before regenerating new ones.

- **Handling external programs (Jess):** Based on the statistics that we collected about the two classes (see tables 1 and 2), Agitar did not comprehensively test the JessManager class, which is a manager for the external Jess rules engine.

**Bottom Line**
Overall the unit tests generated by AgitarOne create a security net for legacy code that can help make developers aware of unintended effects of their modifications. The test cases also provide a framework that can be extended by the developer to achieve better results. The techniques that AgitarOne provides for this are Test data helpers and Assertion helpers. AgitarOne also has its own set of annotations that can be added to the test cases. Although the generated tests are not perfect they provide a very good starting point. Writing all these test cases manually for already existing code is a huge time investment that most projects can't afford.

So, for a large legacy code base without test cases, the only reason for not using the tool is its cost. The project would need to evaluate if the time saved for not writing tests manually is bigger than the acquisition cost. Even if the developer needs to add additional helper classes to enhance the tests, the time saved may still be significant.

Another alternative would be to not use unit testing at all, and just do peer-reviews for all the changes. This variant however seem to be difficult to apply since the legacy code may be unknown and the effects of changes can't be assessed easily. Also in this case the project would need to make a trade-off between the invested time for the reviews and the cost of AgitarOne.

An additional factor in the evaluation of a buy decision, of course is that the tool can be reused in other projects, so that the investment would make sense from an organizational perspective, even if it isn't worthwhile for one project.

If we can continue to use this tool, we will use it to create a regression test suite for our legacy code, and then use it during our implementation stage to make sure that our new code does not break any existing functionality.

# Agitation

AgitarOne's agitation feature automatically generates test input data and enables the developer to observe the code's behavior. In this section, we briefly describe how this feature performs this automatic testing; we describe two experiments that we applied this feature to and the results in terms of bugs that were identified. Finally, we evaluate the feature based on our experience with it.

## How agitation works

The agitation feature starts out by creating instances of classes from the code. Then, it automatically performs method calls with those instances using a wide variety of input data. The input data is based on the types of the class and method parameters. This is called "exercising" or "agitating" the classes and upon completing this process, results are produced. Results include observations and assertions, which are the behavior of the classes and methods it recorded while exercising with different inputs. In addition, it also

shows unexpected behavior – i.e., when a specific input causes one's code to throw an exception, and this exception was not caught. Essentially, the results produced through agitation are facts about the classes and methods, it is then left up to the developer to decide whether the observations are what were intended.

## Experiment description

The objective of our experiments was to evaluate how effective agitation is at catching bugs in an ongoing project and the Othello board game. In these experiments we applied agitation giving some assistance to the input data to the first project and without assistance to the second.

### DWP Experiment

In the first experiment we applied agitation to DWP, which is a project that applies formal methods to reason about properties and behavior of Java programs. This application uses Dijkstra's Weakest Precondition (DWP), where given a desired post-condition and the source code, a pre-condition is generated. This resulting pre-condition represents what the source code must satisfy so that when it executes, its final state satisfies the desired post-condition.

Within the code base of this application, there are two parts in which we applied agitation, a parser and an analyzer. The parser uses Crystal to implement the translation of source code to DWP notation. While the analyzer implements the application of the DWP method to the translated source code.

### Othello Board Game experiment

The second experiment was to apply agitation to the Othello Board game project. This project is an implementation of the board game in which we will agitate all of its classes and see if there are unexpected behavior and errors in the code.

## Experiment Results

### DWP Parser

In agitating the parser class, a *NullPointerException* was identified in the constructor. This uncaught exception is a verified bug.
In order to test that the parser is actually translating the source code properly, we need to use the factory feature. We were able to use an existing random factory object to create input data for the *IAssignmentExpressionNode* type (method parameter). This small and limited test did not reveal any issues about the method implemented within this class. Nonetheless, Agitator observed not-null preconditions on field variables in some methods (*e.g.*, this.x != null). These are useful and valid observations that were then turned into assertions for regression tests.

### DWP Analyzer

We agitated several classes from the analyzer with the following results.

| | |
|---|---|
| **Total Warnings Identified** | 27 |
| **Actual Bugs** | 17 |
| **False Positive** | 10 |
| **Time to run test** | 20 seconds |

Agitator identified 17 instances of uncaught exceptions which are either a null pointer exception or an array index out of bound exception. These are valid bugs that we

somehow overlooked while writing the code. They are all located within a specific part (data repository) of the analyzer. We consider the severity of these bugs to be medium. In addition, there were 10 issues that the tool warned about, but we were not concerned with them because they were not real issues.

Agitator made several observations that we consider valid such as not-null preconditions on field variables in some methods (*e.g.*, this.x != null). These observations can be turned into assertions for regression tests. Agitator also observed some relationships that were valid but too weak. For example, it observed that -100 <= iChildNumber <= 100 while DWP notation allows only up to 3 children nodes. However, we are able to edit this observation and turn it into a stronger assertion.

### Othello Board Game

The results of agitating BoardGame are shown in the following table.

| | |
|---|---|
| **Total Warnings Identified** | 47 |
| **Actual Bugs** | 30 |
| **False Positive** | 17 |
| **Time spent to run test** | 30 seconds |

Agitator identified 30 instances of uncaught exceptions of 8 distinct types (NullPointer, Runtime, AssertionError, NumberFormat, NoSuchElment, ArrayIndexOutOfBounds, and ClassCast). These are valid bugs that were scattered throughout the project. We consider the severity of these bugs to be medium.

The tool was not able to test all classes in the project due to timeout issues, especially for the *AIThread* class. The AgitarOne server timed out when generating the tests because they took too long to compute. The figures in the table could have been higher if we could have agitated the *AIThread* class successfully.

## Evaluation

**Benefits**

- **Unexpected behavior:** The tool uncovers unexpected behavior more effectively than human methods (i.e., manual unit testing) through its ability to test classes and methods with a variety of inputs. For instance, it was able to identify null pointer exceptions in a critical part of the analyzer, and this was very useful because we overlooked that exception when writing those parts.

- **Efficiency:** The tool identified valid observations and possible assertions about the code within a short amount of time. Basically, it identifies what the code actually does rather than what we intended the code to do.

**Limitations**

- **Coverage:** The tool did not have 100% code coverage for classes because it could not generate input data for some methods to test it properly.

- **Learning curve:** For agitation to be useful in most cases, one will have to write factories to assist the agitation process, so that it guides the creation of input data. However, writing good factories requires some upfront work in learning the factories API. Therefore, given more time to use this tool, we could implement some factories to make the most of applying it to a complex code base like Crystal.

**Bottom Line**
The agitation feature provided by AgitarOne gives you the ability to assess the quality of your code with concrete evidence. Through some up front work in developing factories to guide agitation, one can benefit later in testing and catching bugs in one's code. In addition, despite this features limitations, it proves to be more effective over manual unit testing because of the difficulty in coming up with a variety of inputs to test methods thoroughly with unit testing. Thus, if we continue to use this tool, we will create factories for testing our code, and then use it for agitation throughout implementation.

# Code Rules

AgitarOne's code rules can inspect code automatically to ensure compliance with standards and detect many simple errors as well. In this experiment we evaluate how useful are the code rules for existing legacy code bases.

Given that software applications are not usually maintained for their entire lifetimes by the original authors, the code should follow standards and conventions that improve the readability and reduce the possibility of errors.

Code Rules are graded into 4 categories of decreasing severity: error, warning, info, and ignore. The rules are also classified into different groups such as:

- Coding and naming conventions
- Formatting – not selected as default rules, would be useful in an organization setting.
- Metrics – such as complexity, length, and method counts
- Object oriented programming best practices
- Possible bugs – these are important since they may indicate hidden bugs
- Unused code
- Specialized rules (for J2EE, JUnit, and Javadoc)

## Experiment description

The objective of the experiment was to evaluate how useful AgitarOne code rules are at finding improperly written code in the Serendipity code base. In this experiment we used AgitarOne default selection of code rules.

## Experiment result

### Serendipity code base

Although the code base is significant (20 KSLOC), we felt that AgitarOne should not flagged it down too much since we have been assured by indicators such as customer acceptance, test results, and the attention paid to quality. The result from running Code Rules mostly confirmed our *priori* judgment.

There are 34 errors, 506 warnings, and 7 infos. Most of these are of the same categories, however. We will describe notable findings as follows.

**Errors**
All 34 errors are due to 3 code rules.

- **Combining assignment within an if condition**
  For example,
  ```
  if ((error = Native.deleteSensor(svSensorID)) != 0) {
  ```

  These are not actually errors, but can be confusing.

- **A clone method does not call super.clone()**
  Again, these are not actually errors.

- **A class overrides equals() but does not override hashCode()**
  This is a possible source of bug because we may lose data if the class is used as the key in a hash table. Currently, none of these flagged classes are used as hash table keys. Nonetheless, it is a good preventative measure to override their hashCode() methods.

**Warnings**
Most of the warnings are due to unused methods, fields, parameters, and local variables. These dead codes are not bugs, but can create confusion for maintainers. The rest of the warnings belong to the following categories:

- **Exceeding metrics**
  - Too many cases (> 10) in a switch statement
  - Cyclomatic complexity > 15
  - A method makes more than 100 method calls

  These flags indicate a place where we may want to refactor the code, but are not actually bugs. The very complex method that makes more than 100 method calls is a method that creates icons for components in a tool palette. Since there are many components, there are many calls. The methods with high cyclomatic complexity are all user interface handling codes.

- **Using == to compare objects, instead of using equals()**
  Some of these comparisons are not a problem since they compare statically created objects against one another. On the other hand, there are some comparisons that may cause trouble and we should correct them.

- **Interface implementations should be abstract or non-trivial**
  Codes flagged by these simply have empty implementation (do nothing or return null). Some of them are not bugs but some of them are classes that were not finished due to time constraint. All unfinished works that we have known about before running the tool are correctly flagged with this rule.

- **Using StringVar.equals("String literal") instead of "String literal".equals(StringVar)**
  This can be a problem if the string variable is null, resulting in NullPointerException.

**Infos**
All infos are due to empty catch blocks. Some of them have comments which note that the catch block should exist without doing anything other than catching the exceptions. But some of them do not have comments, which may mean the implementation is not finished.

## Crystal

We also ran Code Rules on the Crystal code base to see if we can get the same results. Code Rules issued 5 errors, 367 warnings, and 4 infos for Crystal. Most of the issues are similar to the ones we found in Serendipity's code base, with two notable additional issues:

- **Class inheritance hierarchy deeper than 5**
  This flag indicates that Crystal class hierarchy is very complex, which is probably warranted.
- **Subclassing RuntimeException instead of normal (checked) Exception**
  We don't know enough about Crystal implementation to determine whether this is a mistake or an intentional violation of convention.

### Evaluation

**Benefits**

The tool is fast and does not consume much resource. The resulting output provides us with possible location of bugs, dead code, and areas with high complexity. Most of the possible bugs are easy to fix.

**Limitations**

The false positives rate can be very high if the chosen rules do not match the coding standards followed by the developers. This problem can be "fixed" by turning off that specific rule.

**Bottom Line**

The Code Rules feature provided by AgitarOne is useful for legacy code. The rules can point out not only the possible location of bugs but also where we should focus the effort of refactoring. The dead code and the methods with high complexity should be priority targets for refactoring.

If we can continue to use this tool, we will use it to complement inspection by forcing developers to check against Code Rules before submitting the code for inspection. This automated tool can help increase inspection efficiency.

## Management dashboard

The management dashboard is a view which collects results of tests and shows the summary of the project. Because we did not have data collected over a period of time, we were unable to perform an experiment using this feature. However, because it is a key part of Agitar, we try to evaluate it here.

Features:

- Analyzes the results of JUnit Tests and agitation
- Reports on code-rule violations
- Works with existing Ant build/test scripts or AgitarOne project files
- Automatically associates test classes to project classes
- Assigns test points for JUnit and AgitarOne assertions
- Let's you set targets for testing and coverage metrics
- Shows progress over time
- Sends executive summary email
- Sends customized email reports with each developer's status

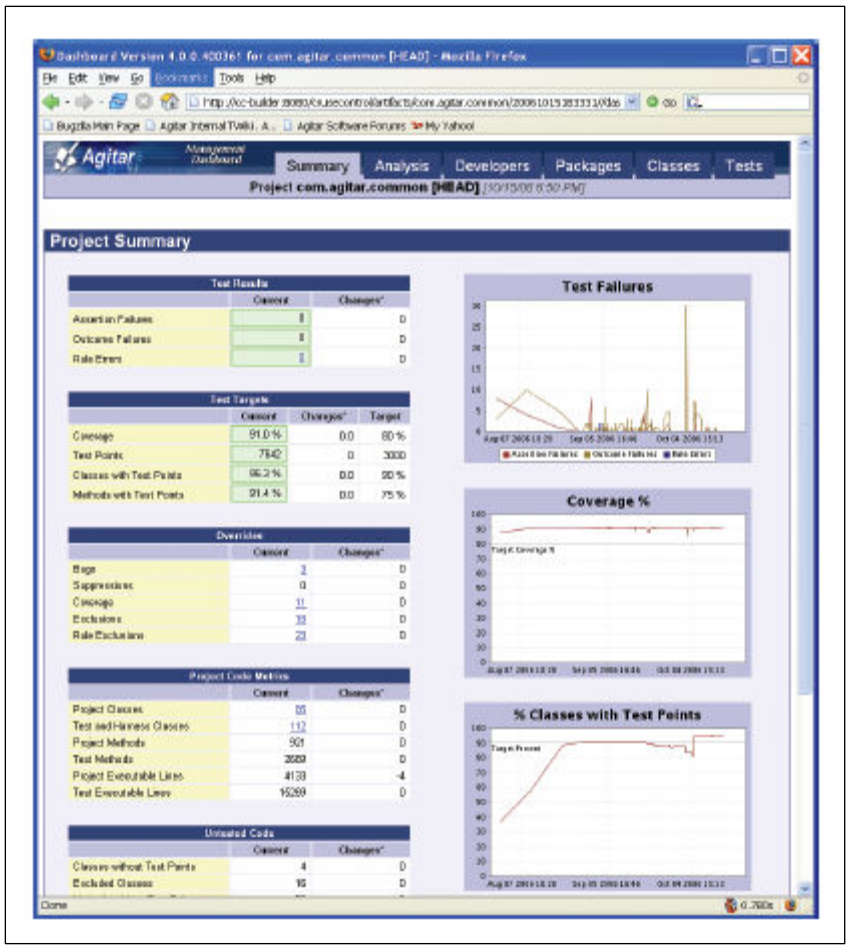


**Figure 1: This is an example the project summary view of the management dashboard[2]**

**Benefits**

- **Useful for project management:** This is because it summarizes the project's testing status at a glance. The project summary shows complexity, usage, coverage, and test status over time.

**Limitations**

- **Server:** It needs a dedicated server to run it.
- **Employee morale:** Manager may use test performance to evaluate employees.

**Bottom Line**

If we can use this tool to manage our projects, we can use it to collect metrics since we are process-oriented.

[2] "Using the Agitar Management Dashboard", Version 4.1, March 2007

# Conclusion

From the evaluation of each of the features we can conclude that AgitarOne provides many benefits for a project.

The test generation feature is especially useful when used in conjunction with a large legacy code base that doesn't have unit tests. It spans a security net that can catch many side effects of refactoring activities. For new development, the Agitation function provides benefits with the intensive test input generation. It helped us to spot some possible problems such as a null pointer exception that could led to an application crash during the production phase. Additional features such as the code rules validation can find further potential problems or inconsistencies. This also can save a lot of manual work during code reviews. The management dashboard is a big plus for projects that want to have good tracking quality assurance. The automated gathering of these metrics adds to accuracy and can largely reduce the time spent to manually elicit this data from the system. Although all these features also have their drawbacks and should not be used blindly, the benefits in our opinion outweigh.